HEWLETT PACKARD

HP-71

Software Developers' Handbook

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Section 21
Forth Utilities

Introduction

SECTION 1

This document is a 'cookbook' for applicati is programmers working with the HP-71. Two goals are envisioned: first to serve as a timesaver, and econd to suggest a measure of consistency among programs written for the HP-71. While there is no iope of addressing all possible applications on the HP-71, common subjects such as user interface, envii inment preservation, and error trapping are discussed. The specifics of each application are left to the pr grammer.

Version Identification

SECTION 2

Any BASIC, BIN, or LEX file which is a) likely to hit a wide market, b) not so trivial as to be 100% perfect, and c) likely to have software written to interface to it, is a candidate for requiring version numbering. Like the mainframe version number (eg: 1BBBB), a version number is useful in identifying the version of a piece of software which may go through several revisions. Service and support personnel may need to know which version of software is in use to help answer questions.

LEX files contain a poll handler which answers the VER$ poll. For instance, the KEYBOARD Lex file returns the string "KBD:A". For more information about the VER$ poll, refer to the *Software Internal Design Specification, Volume I.*

BASIC and BIN files should include a subprogram named VER, that returns the version string. For example:

```
SUB VER(A$) @ A$="001" @ ENDSUB
```

This occupies 34 bytes. A CALL statement may be used to determine the version of the software as follows:

```
CALL VER(A$) IN <file name>
```

Nonexecutable files (eg. DATA) cannot respond to the VER$ poll, or contain a subprogram. If such a file is revised, some method of identifying the version should be provided, such as a dedicated record containing a version number. If a data file is in the HPAF format, a tag in the descriptor block might be used to contain a version number. Section 16 contains a description of the HPAF file format.

Working Environment

SECTION 3

The "working environment" defines the physical environment, hardware and software configuration under which tasks are performed. This "environment" may have varying impact upon software considerations. For instance, if the HP-71 is being built into an instrument as a "front panel", the hardware configuration is likely to remain fixed, with only dedicated software in use. At the other end of the scale, a mechanical application program might be found in a number of different situations, from the classroom to the drafting table to the machine shop. In each of these situations the number or type of peripherals attached to the HP-71 may be different. Software routines which produce reports may, under some conditions, need to be sensitive to varying configurations.

3.1 Printer Assignments

When different printers may be used, a distinction between printer types is desirable. The following subprogram PRTYPE examines the current printer assignment and returns:

A = 0	Where the printer is LCD, *, or there is no HP-IL interface.
A = 1	Where the printer is a 24-column strip printer, or 32-column video interface.
A = 2	For anything else.
D$ =	Assignment string

```
10 SUB PRTYPE(A,D$)
20 ON ERROR GOTO 130
30 A$=PEEK$("2F7AC",1) @ IF BIT(HTD(A$),3) THEN 130
40 RESTORE IO @ PRINT "";
50 A$=PEEK$("2F794",3) @ IF A$="00F" OR A$="FFF" THEN 130
60 A$=A$[3]&A$[2,2]&A$[1,1] @ A=HTD(A$)
70 L=A DIV 1024+1 @ IF A$[2]="00" THEN 130
80 A=BINAND(A,31)+BINAND(A,992) DIV 32/100
90 Q$=PEEK$("2F6DC",2) @ STD
100 IF L>1 THEN D$=STR$(A)&":"&STR$(L) ELSE D$=STR$(A)
110 POKE "2F6DC",Q$
120 GOTO 140
130 A=0 @ D$="*" @ GOTO 160
140 A=DEVAID(D$)
150 IF A=32 OR A=48 THEN A=1 ELSE A=2
160 OFF ERROR @ END SUB
```

PRTYPE provides a non-intrusive examination of the printer assignment. The principal advantage is that output routines can customize themselves to the existing machine configuration without disturbing the configuration or asking the user any questions. Depending upon the result from a call to PRTYPE, the software may choose to send results to the printer, or send a line at a time to the display, waiting for a keystroke between results to avoid hurrying the user.

3.2 Required Modules

In cases where an application pac requires the presence of another module, a test should be made early on to verify that module's presence. This avoids the situation where an application halts at some line in a program with a mainframe error, leaving the user suspended in an environment with little hope of clean departure.

A simple test involves examining the string returned by the VER$ function for that module. For instance, suppose you wish to verify that the MATH module is in the machine:

```
90 IF NOT POS(VER$," MATH:") THEN DISP "No MATH Pac" @ GOTO 990
```

3.3 Card Reader

If a card reader is required, its presence may be detected by examining location 2C014. A non-zero value at this location indicates a card reader is installed:

```
120 IF PEEK$("2C014",1)="0" THEN DISP "No Card Reader"
```

3.4 Memory Requirements

Calculating the amount of memory needed to run an application at any given time can be difficult. One procedure for estimating memory requirements involves a trick:

1) Execute a END ALL and a DESTROY ALL to collapse environments and variables. Purge any key assignments that may be established within the program.

2) Do a MEM, and write this figure down.

3) Run the application, and pause at a place you suspect takes the most memory. Do a MEM again.

The difference between the two results represents the amount of memory used by the program *at that point*. Next compute an overhead figure to accomodate unexpected events, such as interrupt processing, string operations, and so on. This 'fudge factor' is an insurance policy against unexpected program crashes, such as interrupts from other pacs, larger than expected buffer requirements, and so on.

The 'fudge factor' may vary in size from application to application. Actions that take lots of memory include concatenation of large strings, calls to user defined functions: FNA(A,B,D$), calls to other sub-programs, use of IMAGE statements, and open file channels. Some experimentation may be required to determine an appropriate 'fudge factor'. In previous applications, 300 bytes seems to have been a reasonable size.

In cases where a file is to be added to main memory (file:MAIN), a check should be made to ensure that sufficient memory exists prior to creation. Simply putting an error trap around a CREATE and a MEM test afterwards has proven dangerous. Instances have occurred where sufficient memory was available for file creation, but the program crashed immediately thereafter due to lack of scratch memory for normal execution. The amount of available memory for file creation should be equal to the file size plus the 'fudge factor'.

Working Environment

User Environment Preservation

SECTION 4

Preserving the user's environment can be a fairly difficult issue, depending upon the application. In the case where the HP-71 is being used in a dedicated environment, for one purpose only, there may be little need to worry. In cases where an application is being marketed as a general purpose solution, careful preservation of the environment is extremely important. The HP-71 has many settings that control display attributes, math functions, and so on. These settings are 'global' in nature - they affect all programs and actions. In addition, variables are global, so they might be used by the user to store personal information. It is inappropriate to destroy the user's information.

4.1 Variables

The simplest way to preserve user variables is to run the new application in its own subprogram environment. Create a subprogram with the same name as the user would type. For example:

File: AUDIT

```
10 ! AUDIT Copyright (c) LAH Inc., 1984
20 CALL AUDIT
30 SUB AUDIT
40 IF MEM<300 THEN DISP MSG$(24) @ GOTO 9450

 ...

9450 END SUB
```

In this case, the user can press [RUN] when the file pointer indicates the file AUDIT, or he can type CALL AUDIT or RUN AUDIT. When AUDIT terminates, the user's environment is restored, along with his variables.

4.2 Flags

Although the application is running within its own environment, it is vital to remember that system flags (-64 to -1) and user flags (0 to 63) are global - their states are the same regardless of which environment is active. There are two ways to preserve these flags - individually, or as a group. To preserve an individual flag, allocate an integer and store the old value of the flag there until it can be restored. Example:

`100 F5=FLAG(-1,1)`	Set quiet mode, saving old value in F5
`110 S=T/N`	Perform questionable operation
`120 F5=FLAG(-1,F5)`	Restore original value of flag -1.

The system flags are located at 2F6D9 (16 nibbles), and the user flags are located at 2F6E9 (16 nibbles). The IEEE traps are located after the flags at 2F6F9 (5 nibbles). If an application is going to work with a large number of system flags, they can be saved as a group:

`90 DIM F6$[5],F7$[16],F8$[5]`	To save RAM, dimension small strings
`100 F6$=PEEK$("2F6D9",5)`	Save user-settable system flags
`110 F7$=PEEK$("2F6E9",16)`	Save user flags
`120 F8$=PEEK$("2F6F9",5)`	Save IEEE traps

-or-

`100 DIM F9$[37]`	Create on string for all flags
`110 F9$=PEEK$("2F6D9",37)`	Save all flags in the same string

When the program terminates, restore the flags with a poke:

`9940 POKE "2F6D9",F9$`	Restore original flag values

Note that using PEEK and POKE for preserving and setting numerous flags results in a significant code saving over the same procedure using CFLAG and SFLAG. For instance, if an application needs to assert quiet mode and continuous operation, leaving other system settings in their default (power on) settings:

```
200 F3$=PEEK$("2F6D9",5) @ POKE "2F6D9","50000"
```

-instead of-

```
200 F3$=PEEK$("2F6D9",5) @ SFLAG -1 @ CFLAG -2 @ SFLAG -3
210 FOR I=-20 to -4 @ CFLAG I @ NEXT I
```


4.3 I/O Assignments

If the application requires changing I/O assignments, the PRINTER IS and DISPLAY IS assignments may be preserved and restored.

To save:

DISPLAY IS:	`70 D9$=PEEK$("2F78D",7)`
PRINTER IS:	`80 P9$=PEEK$("2F794",7)`
KEYBOARD IS:	`90 K9$=PEEK$("2F79B",7)`

To restore:

DISPLAY IS:	`9400 POKE "2F78D",D9$` `9410 POKE "2F7B1","7" @ RESTORE IO`
PRINTER IS:	`9420 POKE "2F794",P9$`
KEYBOARD IS:	`9440 POKE "2F79B",K9$ @ POKE "2F[illegible]","0"`

Another approach to preserving the printer assignment might include prompting the user for alternate assignment. In the case where having a more "human readable" representation of the printer assignment is desired, use the subprogram PRTYPE (in chapter 2).

4.4 Display Attributes

Display attributes such as WINDOW, DELAY, WIDTH, PWIDTH, and ENDLINE may be preserved and restored. Use PEEK and POKE to preserve these settings.

Address	Length	Description
2F471	4	Window start and length
2F946	4	Scroll and delay rate timer
2F94F	2	Display width
2F958	2	Printer width
2F95A	7	ENDLINE length and characters

(Lowercase mode is system flag -15)

4.5 Alternate Character Set

Characters with ASCII character codes from 128 through 255 may be redefined by the user to represent alternate forms, or letters. If an application needs to define some alternate characters, any existing character definitions should be saved and restored.

```
1100 DIM C$[LEN(CHARSET$)] @ C$=CHARSET$
 ...
9999 CHARSET C$ @ END
```

4.6 [ATTN] Key

The [ATTN] key may be locked out, preventing the user from suspending the program. There are two methods of locking the [ATTN] key: redefining the key and using a POKE statement.

CAUTION

DISABLE: POKE "2F441","F"

ENABLE: POKE "2F441","0"

The POKE statement will prevent the [ATTN] key from suspending a program. In the event of catastrophe, an INIT: 1 will usually bring back the HP-71.

To prevent the [ATTN] key from suspending an executing INPUT statement, use a DEF KEY assignment, eg: DEF KEY "#43",""; . In this case the user's keys file will need to be preserved and restored.

Past experience indicates that if the [ATTN] key is to be locked out, both methods should be used.

4.7 Numeric Settings

The settings that control the format of numbers, FIX, STD, and ENG may be preserved and restored. These settings are controlled by system flags. A quick way to preserve them is with a PEEK:

```
250 F5$=PEEK$("2F6DC",2) @ STD @ A$=STR$(G) @ POKE "2F6DC",F5$
```


4.8 Key Files

An application that redefines the keyboard will have to preserve and restore the user's key definitions. Several existing pacs have dealt with this i ue: Finance, Curve Fit, and Text Editing. In each case, the current keys (if they exist) are kept in a emporary file "USERKEYS". To prevent any chance of a program 'crash' leaving the user suspended w th a redefined keyboard, restrict the duration of redefinition as much as possible. For example:

```
100 I=FLAG(-1,1) @ I1=FLAG( 9,1) ! Set quiet mode, user mode
110 PURGE USERKEYS @ ON ERR R GOTO 130
120 RENAME KEYS TO USERKEYS
130 MERGE PACKEYS @ ON ERRO  GOTO 150 @ POKE "2F441","F"
140 DISP MSG$(16); @ INPUT I$
150 OFF ERROR @ PURGE KEYS   ON ERROR GOTO 170
160 RENAME USERKEYS TO KEYS
170 POKE "2F441","0", @ GOT  'PROCESS'
170 DISP ERRM$ @ GOTO 140
180 'PROCESS': !
```

This routine is useful when entering a strin or responding to hidden key definitions. For instance, with this routine the user could either enter a s ring, or press a previously defined key to branch to another part of the program. This is one instance here key definitions terminated with a colon ':' are very handy. Suppose the following keys are defi ed:

```
DEF KEY '+','add71':
DEF KEY '-','sub71':
DEF KEY 'f0','';          Lock ut USER key
DEF KEY '#50','';         Lock ut [^]
DEF KEY '#51','';         Lock ut [v]
```

If the user presses the [+] key, I$ would take the value add71, and the display would remain unchanged. Likewise, if the user presses the [-] key, I$ ould take the value sub71. If needed, the contents of the display after the prompt can be read with th DISP$ statement.

NOTE

In the above example, the U ER mode key and the keys for the command stack have been 'locked ut'. While each application has different requirements, there may be o e or more keys which should be locked out to provide a 'cleaner' interface. This merits careful examination.

User Environment Preservation

When defining keys for an application, keep in mind that a foreign language might use a different letter for a certain response, so MSG$ should supply the definitions. For example, if you want [Y] to display the word "Yes", use

```
410 DEF KEY MSG$(143061)[1,1],MSG$(143061);
```

instead of:

```
410 DEF KEY "Y","Yes";
```

WARNING

This technique depends on each option having a different letter for each response. When translating an application ensure that each command in a prompt begins with a unique letter!

4.9 Manual Consideration

While an application may preserve global system settings, it is still important to indicate their use in the owner's manual. In the event of a breakdown of the software, the user should be able to recover his environment with help from the manual. Information in the manual should include a list of settings that are changed and a list of any temporary files that are created.

Messages To The User

SECTION 5

Prompts, status messages, and error messages destined for the LCD should be easy to understand and spelled correctly. When shortening a message, do not introduce ambiguity by eliminating too many words. Also, to shorten individual words, omit as few letters as possible. Try to avoid cryptic messages. In addition, messages should fit within the display window.

The following is a guideline for messages in the display:

1) A question mark implies that some response is required:

   If a cursor is present, the entry is terminated with the [ENDLINE] keystroke (such as file name entry).

   If no cursor is present, the first letter of each word denotes the appropriate key to press. The first letter should be capitalized, the rest should be in lowercase.

2) If a long operation is in progress, a status message is suggested. No response is required. For long calculations, use "Working...".

3) Use mainframe messages as much as possible, or use similar internal words.

Some examples:

Display	
`Working...`	Status message
`Loading...`	Status message
`Data Edit Fit Quit?`	Immediate execute menu (no cursor)
`File name?`	Prompt with cursor
`WARNING: Low Voltage`	Warning message

`WRN: Underflow Found`	Warning message
`ERROR: Zero Tolerance`	Error message
`ERR: Insufficient Mem`	Error message
`Total= 247.232`	Result

If an application is destined for a world market, MSG$ should be used to generate all messages (and all comparison strings for incoming messages.) The MSG$ keyword fetches messages from a LEX file table, allowing them to be accessed by number. More importantly, MSG$ performs a translating function. For example, the message

```
Working...
```

could be displayed with:

```
120 DISP MSG$(104036)
```

so that a localized Spanish language version of the application pac would display:

```
Trabajando...
```


Waiting For The User: KEYWAIT$

SECTION 6

There are many circumstances where the HP-71 is doing little more than sitting, waiting for a keystroke. During these times, the machine is still awake, consuming battery power, while accomplishing little for the user. A keyword called KEYWAIT$ is available, and presents some unique opportunities. KEYWAIT$ places the HP-71 into a low-power state until a key is pressed, then returns that key in the same format as KEY$. IMPORTANT: If the attention key is not disabled, KEYWAIT$ will return "#43", but the machine will still pause.

Example:

```
350 K$=KEY$ @ IF K$="" THEN 350
```

is replaced by:

```
350 K$=KEYWAIT$
```

The key buffer can contain up to 15 keys or keystroke combinations. The format in which the key data is returned is the same as that for KEY$. The string returned for a given key is determined as follows:

- If there is a single ASCII character that uniquely identifies the key, KEYWAIT$ returns this character. For example, Q identifies the [Q] key and q identifies the [g]-shifted [Q] key.
- If the key is an [f]-shifted or [g]-shifted key, and the key's primary function is uniquely identified by a single ASCII character, then KEYWAIT$ returns a two-character string. This string consists of f or g followed by the corresponding primary character. For example, g0 is the [g]-shift of the [0] key.
- If neither of the above apply, KEYWAIT$ returns # followed by the decimal numbered key code for that key. For example, KEYWAIT$ returns "#46" for the [RUN] key.

The LC statement does not affect the returned string.

Option Selection

SECTION 7

At first appearance, the 22-character display on the HP-71 might seem to be an obstacle to creating friendly menus. Actually the architecture of the HP-71 provides for several possibilities. Regardless of the specific application, option selection in a handheld/portable environment should be reduced to the bare minimum of keystrokes. Prompts in the display should be as legible as possible.

7.1 Command Entries

When an application is command driven, (the Text Editor, for example), consistency in movement between states becomes of paramount importance. If a command is defined in some places as a handy 'escape' key, it should work the same way at all times. Entries should be case independent if possible, so that commands work regardless of the case of the entry. If possible, build the display prompt with some 'clue' as to the state of the program. For instance, the Text Editor uses different prompts between command and editing levels.

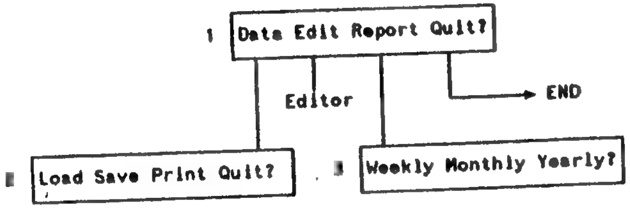
Some examples:

Input/Result

`Recording option?`	**Command prompt**
`Level 5: Command?`	**Command prompt w/ status**

7.2 Immediate Execute Menus

When a tree-structure of options is used, immediate execute menus should be used. This reduces the selection of an option to a single keystroke. For example:



In each display, the key choices are indicated by the capital letters. Pressing the [D] key in the first box leads to box 2, and so on. In this application, Quit is the escape mechanism. In box 3, the [Q] key should also be active to enable a return to box 1, so as to be consistent with the other menus. An example implementation of box 1 looks like:

```
270 DISP "Data Edit Report Quit?"
280 P=POS("DERQ",UPRC$(KEYWAIT$))+1
290 ON P GOTO 270,'DTA','EDT','RPT','QUIT'
```

Notice that this handles an unusual circumstance with little extra effort. Suppose the user presses the attention key and suspends the program. When the user presses [f] [CONT], execution of line 280 will proceed with the result of KEYWAIT$ being "#43". The POS command will return zero, causing the branch to line 270. This restores the prompt in the display, so the users may continue without confusion as to where they are in the application.

CAUTION

Some shifted keystrokes will return "f" or "g" as the first character of the result of KEYWAIT$. If "F" or "G" are allowable keys, they should appear last in the match string to avoid possible input errors.

7.3 Fixed Option Menus

Configuration of a device or a set of preconditions for a calculation can be reduced to a few keystrokes by presenting a two-dimensional picture of the options. This 'picture' would contain all available, fixed options.

For instance, suppose a multimeter is being configured for an experiment. There are four settings to be made: the type of measurement, accuracy of the measurement, choice of input channels, and data rate. Each setting has a number of different options. A common approach is to prompt for each setting. A faster method is to present a menu that can be scrolled by the user.

CURRENT OPTION	AVAILABLE OPTIONS				
MEASURE: Ohms	hms	Resistance	Current		
ACCURACY: 3 Digits	Digits	4 Digits	Digits		
INPUT: Front	ront	Back			
DATA RATE(Pt/min) 100	00	200	500	1000	5000

Define the vertical arrow keys [^] & [v] (#50 and #51) to scroll in wrap-around fashion between MEASURE, ACCURACY, INPUT, and DATA RATE. Define the horizontal arrow keys [<] & [>] (#47 and #48) to scroll in wrap-around fashion between the options at the current setting.

Selection of an option could be made simply by leaving the option in the display, or requiring a keystroke to select the option. Another key would be used to exit the menu.

The user can now examine all four settings and exit the menu with four keystrokes. If the user wishes to alter just one option, the maximum number of keystrokes ever needed would be 7, including the exit from the menu!

Option Selection

Suppose the user wishes to change the accuracy from 3 digits to 4 digits. In this example, a horizontal option is selected merely by being in the display, and the [ENDLINE] key exits the menu. The sequences on the left and right yield identical results. The right column shows the user taking advantage of the wrap around selection of choices.

`MEASURE: Ohms`	`MEASURE: Ohms`
`[v]  Goto ACCURACY setting`	`[v] Goto ACCURACY setting`
`ACCURACY: 3 Digits`	`ACCURACY: 3 Digits`
`[>]  Select 4 Digits option`	`[<]  Select 5 Digits option`
`ACCURACY: 4 Digits`	`ACCURACY: 5 Digits`
`[>]  Select 5 Digits option`	`[ENDLINE]  Exit menu`
`ACCURACY: 5 Digits`	`Press any key to begin`
`[ENDLINE]  Exit menu`	
`Press any key to begin`	
`TOTAL:  Four keystrokes`	`Three keystrokes`

When the user returns to the menu again, the 'accuracy' will be 5 digits.

Note that an implementation of this technique can be 'smart'. For instance, suppose that the maximum data rate for 5 digit accuracy is 500 readings per minute. If the user enters the data rate option, only the 100, 200, and 500 options are presented. If the user enters the accuracy option while the data rate is 2000, only the 3 digit accuracy option is visible. Possibilities abound!

Help!

SECTION 8

If an application uses a variety of commands that must be obtained from the manual, or a possibly missing keyboard overlay, a help file deserves consideration. There are many ways to implement a help function - each application's needs will be different. The following routine suggests one method. This routine reads lines from a text file, making foreign language translation possible. In this example, the help routine is activated by pressing the [?] key when a menu prompt is in the display.

```
180 DISP "Data Edit Report Quit?"
190 K$=UPRC$(KEYWAIT$) @ F=POS("DERQ?",K$)
200 ON F+1 GOTO 180,'DATA','EDT','RPT','QUIT','HELP'
...
1000 'HELP': A=1 @ N=16 ! N=Number of records in help file
1010 ASSIGN #89 TO "HELPTEXT"
1020 READ #89,A;Z$ @ DISPLAY Z$ @ K$=UPRC$(KEYWAIT$)
1030 IF K$="Q" THEN ASSIGN #89 TO * @ GOTO 180
1040 IF K$="#50" THEN A=MOD(A-2,N)+1
1050 IF K$="#51" THEN A=MOD(A,N)+1
1060 GOTO 1020
```

Note that if the application is driven from specific keys on the keyboard, the help routine above may be extended. For instance, if pressing the [\/] key *always* triggers a specific action, such as computing an interest rate, the help routine could respond to [W]. In the above example, if the 5th record in the help file describes the interest rate calculation, the following line of code could be added:

```
1055 IF K$="W" THEN A=5
```

Clearly there are many possibilities for help files beyond this example. Experimentation is encouraged. If an application can be easily run without a manual and keyboard overlay, using just the built-in help commands, the user will spend more time thinking about the task at hand, rather than computer science problems.

Input Routines

SECTION 9

Standard input routines for an application reduce the chance of error, add consistency to a program, and make the programmer's job easier. The following input routines are suggested for normal entry of numbers and strings.

9.1 Cursor Control

When the user is editing an entry, the cursor may not be placed over characters in the display buffer that were written when the cursor was 'off'. This technique is used in the implementation of the INPUT statement, where the cursor may not override the prompt. This may be used in the construction of custom input sequences.

To create non-editable characters in the display, send a 'cursor off' sequence before the characters that you wish to protect, then send the 'cursor on' sequence. The 'cursor on' sequence is CHR$(27)&">", and the 'cursor off' sequence is CHR$(27)&"<". The cursor control characters are not counted in the 96 character length of the display buffer.

Section 9.6 has an example of protected field entry which uses cursor control sequences to enter a date.

9.2 Numeric Entry

This routine accepts a single number from the keyboard. For non-real data types, a declaration of the type is suggested at the start of the program. Note that if the type of incoming data is invalid, such as a string "=&$%", the system error message will be displayed and the user will be prompted again. Line 1300 requires a quantity that is greater than or equal to 1, and is not a NaN or Inf.

```
100  REAL Q

1270 ON ERROR GOTO 1290
1280 INPUT "Quantity?";Q @ OFF ERROR @ GOTO 1300
1290 DISP ERRM$ @ GOTO 1280
1300 IF Q<?1 THEN DISP "ERR: Invalid Quantity" @ GOTO 1270
```

If the application is going to have foreign language capability, the routine looks a little different. For the following example, assume that message 5023 reads "Enter quantity?"

```
1270 ON ERROR GOTO 1300
1280 INPUT "",CHR$(27)&"<"&MSG$(5023)&CHR$(27)&">";Q
1290 OFF ERROR @ GOTO 1310
1300 DISP ERRM$ @ GOTO 1280
1310 IF Q<?1 THEN DISP MSG$(5024) @ GOTO 1270
```

The escape codes in line 1280 are used to turn off the cursor while displaying the prompt, and then turn on the cursor again. The trick is that when the user is editing the response, the cursor cannot be positioned over a character that is written to the display when the cursor is off. All of this is done so that the user can pause the program with the [ATTN] key, press [f] [CONT], and get the prompt restored in the display.

If a default answer is going to appear, say 24, line 1280 would look like this:

```
1280 INPUT "",CHR$(27)&"<"&MSG$(5023)&CHR$(27)&">24";Q
```

```
Enter quantity?24
               ↑
Leftmost possible position for cursor
```

NOTE

For convenience and ROM savings, the escape codes can be imbedded in the MSG$ message table itself.

9.3 Numeric Entry With Option

The Curve Fitting ROM has a situation where the user may select a single row from an array for evaluation, or all rows as a group. A hybrid input module was devised which would let the user enter the character [A] to evaluate all rows, or enter the number of an individual row. Pressing the [A] key results in immediate printing of all rows, with no [ENDLINE] keystroke required.

The following code (altered slightly for the example) was used:

```
1000 DISP "Row # (or All)";CHR$(27);">" @ Q$=KEYWAIT$
1010 IF KEYWAIT$="#43" THEN 1000
1020 DISP CHR$(27)&"<"; @ IF UPRC$(Q$)="A" THEN 1070
1030 PUT Q$ @ ON ERROR GOTO 1050
1040 INPUT "";A @ OFF ERROR @ GOTO 1060
1050 DISP ERRM$ @ OFF ERROR @ GOTO 1000
1060 DISP A @ STOP
1070 DISP "ALL" @ STOP
```

Line 1000 displays the prompt, turns the cursor on, and waits for a key. If the [ATTN] key is returned, indicating that the program was suspended, the prompt is displayed again.

Line 1020 turns off the cursor, and tests for the character "A". If the character is an "A", the program branches to the module for printing all rows in the array.

Line 1030 places the character back in the input buffer, and uses an INPUT statement to obtain an individual row number. If an error is encountered, it is important to rebuild the prompt, so the error trap branches back to line 1000.

Experimentation with this technique is encouraged – it may be useful in simplifying user interfaces and/or reducing the number of questions put to the user.

9.4 String Entry

Entry of strings is similar to that of numbers:

```
100 DIM F$[8]

...

2170 ON ERROR GOTO 2190
2180 INPUT "LOAD: File name?";F$ @ OFF ERROR @ GOTO 2200
2190 DISP ERRM$ @ GOTO 2180
2200 IF LEN(F$)=0 THEN DISP "Invalid Filespec" @ GOTO 2170
```

Or, for foreign languages:

```
2170 ON ERROR GOTO 2190
2180 INPUT "",CHR$(27)&"<"&MSG$(5029)&CHR$(27)&">";F$
2190.OFF ERROR @ GOTO 2210
2200 DISP ERRM$ @ GOTO 2180
2210 IF F$="" THEN DISP MSG$(5022) @ GOTO 2170
```

Again, the same technique with the cursor used in the numeric input module is used for string entry. If there is a default answer, it may be included after the cursor-on command.

9.5 Yes or No?

Answering yes/no questions should require just one keystroke. If an application has many of these questions, a function may be created to simplify the process:

```
160 DEF FNY(Q$)
170 DISP Q$ @ I=POS("NY",UPRC$(KEYWAIT$[1,1]))-1
180 IF I<0 THEN 170
190 FNY=I @ END DEF
```

If the user suspends the program during this function with [ATTN], and then restarts, the prompt will be restored to the display.

NOTE

For foreign language purposes, line 170 might read:

```
170 DISP Q$ @ I=POS(MSG$(13067),UPRC$(KEYWAIT$[1,1]))-1
```

9.6 Protected Field Entry

Some entries, such as dates may require either a limited number of characters, or a specific number of characters. This may be made more apparent to the user with the use of *protected field* data entry. For example, suppose the user is going to enter a date. A protected field template may be constructed to indicate the number of required characters, as well as the sequence of month, day, and year fields:

Date?mm/dd/yy	The user is prompted for the date.

The following routine may be used to set up the date template:

```
10 OPTION BASE 1
20 DIM I$[110]
30 C$=CHR$(27)
40 I$=C$&"<Date?"&C$&">mm"&C$&"</"&C$&">dd"
50 I$=I$&C$&"</"&C$&">yy"&C$&"<"
60 I$[110]=" "
70 INPUT "",I$;D$
80 DISP D$
```

Notice that I$ is 110 characters long. The cursor on and cursor off sequences are not included in the 96 character count for the limit of the display buffer.

The user may type over the characters mm, dd, and yy *only*. The HP-71 will beep after the last y is edited, and the cursor will remain in that position. The cursor keys may still be used to edit the entry.

INPUT Alternative: INLINE

SECTION 10

The INLINE keyword (available in the "CUSTUTIL" LEX file) adds extended cursor control and extended termination capability for user input. Editors, menus, protected fields, and custom entry sequences are possible with INLINE.

Syntax

```
INLINE I$,L1,C1,T$,V1[,V2[,V3]]
```

Parameters

input string — The *input string* (I$ in the example) will appear in the display. Cursor control characters may be imbedded to control which characters may be edited by the user.

first character — The *first character* (L1 in the example) is the index to which character in the *input string* will appear in the leftmost position of the LCD window. For instance, if the *first character* is 3, the third character of I$ would appear in LCD position 1.

cursor start — The *cursor start* parameter (C1 in the example) specifies the starting location and type of the cursor. A negative value specifies the insert cursor. The expression must round to X, such that 1<=|X|<=96.

terminator string — The *terminator string* (T$ in the example) specifies which keys may terminate input. Normally, only the [ENDLINE] key will terminate input from the keyboard (such as with the INPUT statement). The INLINE keyword uses the *terminator string* to extend termination to a specified list of keys. Keys are specified by their physical key code, such as #43 for the [ATTN] key. Keys are numbered in row-major order, from 1 to 56. For f-shifted keys, add 56; for g-shifted keys, add 112. For instance, to allow termination with the [ENDLINE] key and the vertical arrow keys, the terminator string would be "#38#50#51".

terminator variable — Upon termination of INLINE execution, the *terminator variable* (V1 in the example) contains a number indicating which key the user pressed to terminate input. If the key pressed was the second in the *terminator string* list, the *terminator variable* will contain 2.

OPTIONAL PARAMETERS — The following parameters are optional, and need not be used.

cursor position variable — The *cursor position variable* (V2 in the example) contains the final cursor position and type. A negative value indicates the insert cursor.

window position variable — The window position variable (V3 in the example) contains a number indicating which character was in LCD position 1 when INLINE terminated execution.

NOTE

The values returned in the *cursor position variable* and the *window position variable* are affected by the WINDOW settings. For more information, refer to the HP-71 Reference Manual's discussion of the WINDOW statement.

INLINE is a statement that extends the capability given in the HP-71's INPUT statement and KEY$ statement. INLINE allows you to specify:

- The prompt string.
- The number of prompt string characters to be scrolled off the left side of the display.
- Where in the display the cursor is to come up flashing.
- The type of cursor to appear (insert or replace).

INLINE allows the user to press any combination of keys for input and editing, just like the INPUT statement. While INPUT terminates execution only when specific keys are pressed (such as [ENDLINE]), any number of different keys can be defined to terminate INLINE execution. When one of these terminating keys is pressed, INLINE returns a number that indicates which key caused termination. INLINE will optionally return additional values indicating the cursor position/type and number of characters scrolled off the left side of the display on exit.

For increased customization, the input string may contain cursor on and cursor off characters to make certain portions of the string are non-editable. For more on cursor control, see sections 9.1 and 9.6 of this document.

There are three additional limitations placed on the input parameters for *first character*, and *cursor start*:

1) If *first character* is greater than *cursor start*, then *first character* is set equal to *cursor start*.

2) *first character* is limited to 97-WINDOWsize.

3) If *cursor start* exceeds *first character* + WINDOWsize, then the specified *cursor start* takes precedence, and the *first character* is incremented until the *cursor start* character appears in the display window.

For example:

```
INLINE A$,91,80,T$,A
```

According to #1 above, *first character* becomes 80, instead of 91. Then, according to #2 above, *first character* is further reduced to 75 (assuming the default WINDOWsize of 22).

To illustrate #3 above:

```
INLINE A$,60,95,T$,A
```


In order to get character #95 in the display window, character #74 is put in LCD position 1.

Example

The following is an example illustrating the use of protected fields (non-editable characters) in the *input string*:

C$ = default input string

E$ = escape character: CHR$(27)

```
INLINE E$&"<Enter Name "&1 $&")"&C$,2,1,"#38#50#51",A,B,C
```

In this example the user cannot back the cursor up over the prompt since the cursor was turned off. However, they can edit the default input string since the cursor was turned back on. The replace cursor will come up on the first "readable" character, that is, the first character of C$. The first character of the input string will be scrolled off the left side of the display - this was specified by the *first character* parameter.

INLINE will terminate on any of three keys: [ENDLINE], [up-arrow], and [down-arrow]. If [down-arrow] is pressed, A will be 3 on exit. If the user typed in a five character name before pressing a terminator key (assuming no backspaces), B will be 17 on exit (the cursor originally came up on the 12th character and was advanced five positions), and C will be 2.

Note that the *cursor start* argument "counts" readable characters only. Also, DISP$ "sees" readable characters only, so that a DISP$ done in the above example returns only the user input (including the default input), not the prompt itself.

Note that the cursor position argument and the value returned in the first optional variable do not operate exactly the same way. The cursor position argument counts readable characters only, whereas the value returned in B (in the example above) reflects the *total* number of characters in the "free portion" of the display, readable and non-readable.

Also note that because of unreadable characters in the display, the above example is not affected by limitation (1) on the previous page. Even though the *first character* appears to be bigger than *cursor start*, because of unreadable characters in the display, *cursor start* actually designates character 12

SECTION
11

File Name Verification

11.1 File Names For Loading

An applications program may wish to verify the name of a file or subprogram that has been entered by the user. The following routine is useful for trimming unneeded spaces and detecting invalid characters in a file name F$ prior to loading data.

```
7200 IF F$[1,1]=" " THEN F$=F$[2] @ GOTO 7200
7210 I=LEN(F$) @ IF F$[I]=" " THEN F$[I]="" @ GOTO 7210
7220 J=POS(F$,":") @ IF J=1 OR POS(F$," ") OR NOT I THEN 7230
7225 GOTO 7240
7230 DISP "Invalid Filespec" @ RETURN
7240 DISP "Loading..." @ ON ERROR GOTO 7260 @ IF J THEN 7270
7250 Q$=ADDR$(F$) @ GOTO 7270
7260 DISP "ERROR: File Not Found" @ RETURN
7270 ASSIGN #1 TO F$ .....
```

Line 7200 strips leading blanks, and line 7210 strips trailing blanks, leaving the length of the string F$ in I. This will be used later. These lines may be replaced with the keyword TRIM$ (described elsewhere in this document) as follows:

```
7200 F$=TRIM$(F$) @ I=LEN(F$)
```

In line 7220 the variable J takes the position, if any, of a colon. Three tests follow, each of which would indicate an invalid name. The first is the presence of an embedded space. The operating system will only use the characters in front of the space, possibly confusing the user. The first test looks for a null name before a device specifier. The second test rejects a name with an embedded space, even if an experienced user understands the implications. The third test is obvious - if there are no characters in the name there is no file, right? Mostly. There is a bug in an early release version (1BBBB) of the mainframe code that can damage the file chain if a file is accessed in :MAIN with a null name. *IT IS IMPORTANT TO MAKE THIS TEST!!!*

NOTE

This precludes one situation: that where a user wishes to load a file from :LOOP. If the HP-71 is not a system controller, a different procedure will be needed.

The variable I is used again in line 7240 to decide if a file contains any device specifier. If no device specifier is present, the file will have to be in RAM or in a port.

WARNING

> An ASSIGN # statement will create a null length data file in main memory if the file does not exist and no device specifier (":MAIN") is in the name string F$. If there is a colon in F$, there is no danger of creating an empty file.

In order to prevent the creation of an empty file, the ADDR$ function is used in line 7250 to verify the file's existence. Line 7250 actually plays a dual purpose. First, it parses the string F$, and will yield an error if there are any strange characters present. Secondly, if the file is not in memory, an error will occur. Both errors result in a return with an error message in line 7260.

Again, the use of MSG$ is encouraged in place of fixed error messages and prompts.

11.2 File Names For Saving

The following routine is useful for checking file names when saving to a data file. It bears much similarity to the routine used for loading. The routine assumes the file name in F$, the desired number of records is in R, and that the number of bytes per record is in N. Note that 1940 and 1950 can be replaced with TRIM$, as shown earlier.

```
1940 IF F$[1,1]=" " THEN F [1,1]="" @ GOTO 1940
1950 I=LEN(F$) @ IF F$[I]=  " THEN F$[I]="" @ GOTO 1950
1960 J=POS(F$,":") @ IF J=  OR NOT I OR POS(F$," ") THEN
     1970 ELSE 1980
1970 DISP "Invalid Filespe " @ RETURN
1980 DISP "Saving..." @ ON ERROR GOTO 2000
1990 CREATE DATA F$,N,R   SSIGN #1 TO F$ @ OFF ERROR @ GOTO
     2060
2000 IF ERRN#59 OR ERRN#10 9 OR ERRN #255030 OR ERRN #255158
     THEN 2050
2010 IF NOT FNY("Overwrite file (Y/N)?") THEN RETURN
2020 DISP "Saving..." @ PU GE F$
2030 ON ERROR GOTO 2050
2040 CREATE DATA F$,N,R @ SSIGN #1 TO F$ @ OFF ERROR @ GOTO
     2060
2050 OFF ERROR @ DISP ERRM  @ RETURN
2060 IF MEM<300 AND NOT J  HEN PURGE F$ @ DISP "Insufficient
     Memory" @ RETURN
2070 ...
```

This routine accounts for null files, duplicate files in both ram and on a device, and for insufficient memory in either ram or a device. The routine FNY may be found in the chapter "Input Routines". FNY returns a one for yes and zero for no. Notice the offsets used with ERRN that account for foreign language localization. Further information on errors may be found in the chapter "Error Messages". Error 59 is the mainframe error for "File Exists", as is 255030 for HP-IL. Errors 1059 and 255158 account for localization of the "File Exists" error.

11.3 Names Of Subprograms

Verifying the name of a subprogram for existence is similar to the system used for checking data file names. First, the name is checked for valid characters with ADDR$, and then a dummy call is made with intentionally mismatched parameters. The resulting error message will either indicate that the subprogram is not present, or that it is there, but the parameters do not match the test. This routine assumes the subprogram name in A1$ and the file name containing the subprogram in A2$. REMEMBER: the subprogram name can be the same as a file name!

```
3020 ON ERROR GOTO 3030 @ 0$=ADDR$(A1$) @ GOTO 3040
3030 OFF ERROR @ IF ERRN=58 OR ERRN=1058 THEN DISP ERRM$
     @ RETURN
3040 ON ERROR GOTO 3050 @ 0$=ADDR$(A2$) @ GOTO 3060
3050 DISP ERRM$ @ RETURN
3060 ON ERROR GOTO 3070 @ CALL A1$(NaN,NaN,NaN,NaN,NaN,NaN)
     IN A2$
3070 OFF ERROR @ IF ERRN=36 OR ERRN=1036 THEN 3090
3080 DISP "ERROR: ";ERRM$ @ RETURN
3090 ...
```


Output Routines

SECTION 12

Output routines on the HP-71 may take a wide variety of forms, using everything from the 22 characters in the display to 80 column printers. Regardless of the specific form selected, it is vital to insure that the user is able to view the entire result, with all relevant digits of the mantissa and (if applicable) the entire exponent. Further, there should be no time pressure on the user.

12.1 Configuration And Data Volume

Output routines should be sensitive to both the volume of data to be presented to the user and the system configuration. If varying configurations are anticipated, multiple output routines are suggested to maximize legibility of the results and usability of the software.

Results best expressed in tabular form may need one routine for the LCD or strip printer, and another for wide output devices.

Unless specific configurations are going to be used, use of specific printer features must be evaluated with care. If an output routine depends on such features as vertical half spacing (for superscripts and subscripts), the application will not run with conventional printers, such as an HP82905B. Conventional printer features such as form feed capability are generally acceptable. When in doubt, check the capability of several possible target printers for common features. The subprogram PRTYPE can be used to determine what class of printer is assigned.

NOTE

> Output routines should use PRINT statements, while message routines (such as prompts, warnings and errors) use DISP statements. This will insure that the user's PRINTER IS assignment will route the output to the desired location.

The following table may be used to help select a suitable output routine given varying results from PRTYPE:

PRTYPE	REPORT	WAIT?
0	Narrow	Yes
1	Narrow	No
2	Wide	No

12.2 Some HP Printer Features

For reference, the following table contains a listing of common printer features in the HP product line, and the escape sequences that enable them.

OPERATION	FEATURE	ESCAPE SEQUENCE	PRINTER
CR		CHR$(13)	1,2,3,4,5
Formfeed		CHR$(12)	1,2,3,4
Linefeed		CHR$(10)	1,2,3,4
Backspace		CHR$(8)	1,2,3,4
Vertical	6 L/in	ESC &l6D	1,2,3,4
Spacing	8 L/in	ESC &l8D	1,2,3,4
Perforation	On	ESC &l1L	1,2,3,4
Skip	Off	ESC &L0l	1,2,3,4
Select	Normal	ESC &k0S	1,2,3,4,5
Print	Expanded	ESC &k1S	1,2,3,4,5
Mode	Compressed	ESC &k2S	1,2,3,4
	Comp, Exp	ESC &k3S	1,4
	Emphasized	ESC &k9S	1,4
Underlining	On	ESC &dD	3,4
Underlining	Off	ESC &d@	3,4

Printers: 1=HP82905B 2=HP2671 3=HP2631 4=ThinkJet 5=HP82162A

12.3 Multiple Results In The LCD

Results presented in the LCD are especially vulnerable to being lost or forgotten. Since the user may at any time answer the phone, sneeze, or for some reason look away from the machine, a result must be held in the LCD until reciept of the information is acknowledged. A simple way to do this is to call KEYWAIT$, and then continue.

If a long string of results is anticipated, a method of scrolling back and forth through the results is suggested, along with an escape method. The following routine assumes that the results are in an array A, with [illegible] answers, and their titles in a message file from positions 17 to 25. The LEX ID of the message file is 12.

```
2000 N=17 @ D$=PEEK$("2F946",4) @ DELAY 0,0
     ! Save DELAY & SCROLL
2010 DISP MSG$(12000+N);A(N-16)
2020 Q$=UPRC$(KEYWAIT$)
2030 IF Q$#"#38" AND Q$#"#51" THEN 2050
2040 IF N=25 THEN BEEP @ GOTO 2020 ELSE N=N+1 @ GOTO 2010
2050 IF Q$#"#50" THEN 2070
2060 IF N=17 THEN BEEP @ GOTO 2020 ELSE N=N-1 @ GOTO 2010
2070 IF Q$="#162" THEN N=17 @ GOTO 2010
2080 IF Q$="#163" THEN N=25 @ GOTO 2010
2090 IF Q$#"Q" THEN 2020
2100 POKE "2F946",D$ @ RETURN ! Restore DELAY and SCROLL
```

The routine will advance to the next result when either the [v] or the [ENDLINE] keys are pressed. If the [^] key is pressed, the previous result will appear. The [g][^] and [g][v] keys go to the first and last results. The [Q] key exits the routine. If the user attempts to go beyond out of range, a beep sounds.

The use of KEYWAIT$ can go even further in the case of a large table that has been generated. Suppose the program creates a table of results, and the user may only be interested in a subset of the results. One way to address this issue is to ask the user for the location in the table that he wishes to view. Another scheme might be to place the user "in the table", and let him move about with the arrow keys in a two-dimensional version of the routine presented above.

12.4 Large Results In The LCD

If a result is simply too large to fit within 22 characters, scrolling the display is the last resort. The best way to implement this is to preserve the display, set DELAY 9,9 and call KEYWAIT$. The following routine illustrates the technique:

```
10 DIM A$[100]
20 A$="LKJADLKJDSFOGAFBAGESFIJLEWLNDSVJNOIJ"
30 D$=PEEK$("2F946",4) @ DELAY 9,9
40 DISP "Name: ";A$ @ Q$=KEYWAIT$ @ POKE "2F946",D$
```

Another approach to the scrolling technique "windows" the title:

```
10 DIM A$[132]
20 A$="LJKSDAFLJKSDLJKSDFLJUNKFLJKSDFLJKKJSDKJH"
30 D$=PEEK$("2F946",4) @ DELAY 9,9
40 DISP "Name: " @ WINDOW 7 @ DISP A$
50 Q$=KEYWAIT$ @ WINDOW 1 @ POKE "2F946",D$
```

12.5 Numeric Formatting

Numbers that occupy a very large dynamic range (say, a hundred orders of magnitude) will present a challenge when presenting results in the LCD. If the title for the result is very small, there may be room in the display for both the title and the number as displayed in STD format. If there is doubt about available room, an IMAGE statement is suggested. The disadvantage of the IMAGE statement is that the user's display digit setting is overridden.

Internal Calculations

SECTION 13

13.1 Changing Array Sizes

The size of an array may be changed with a new DIM statement. This can only be done in the originating environment. Data is stored in row major order and is not zeroed out during redimension. The following paragraphs address techniques for changing the size of arrays. The examples use an array A with R rows and C columns. The array is of type REAL, and a 300 byte 'fudge factor' is used. Variables I and J are scratch integers, and the array is in OPTION BASE 1.

13.2 Adding And Deleting Rows

Add a new, empty row at N:

```
1000 IF MEM-C*8<300 THEN DISP "Insufficient Mem" @ RETURN
1010 IF N<1 OR N>R+1 THEN DISP "Nonexistent Row" @ RETURN
1020 DISP "Working..." @ R=R+1 @ DIM A(R,C) @ IF N=R THEN RETURN
1030 FOR I=R TO N+1 STEP -1 @ FOR J=1 TO C
1040 A(I,J)=A(I-1,J) @ NEXT J @ NEXT I
1050 FOR I=1 TO C @ A(N,I)=0 @ NEXT I @ RETURN
```

Delete a row at N:

```
1000 IF N<1 OR N>R THEN DISP "Nonexistent Row" @ RETURN
1010 IF R=1 THEN DISP "ERROR: You Need 1 Row" @ RETURN
1020 DISP "Working..." @ IF N=R THEN 1050
1030 FOR I=N TO R-1 @ FOR J=1 TO C
1040 A(I,J)=A(I+1,J) @ NEXT J @ NEXT I
1050 R=R-1 @ DIM A(R,C) @ RETURN
```

13.3 Adding And Deleting Columns

Add a column at N. The data will be scrambled after the DIM is executed so a shuffle must occur. Data is moved from positions at T8,T9 to new locations G8,G9. The pattern works backwards, shifting data up to fill the new top locations, straightening out the columns, and setting the new column to zero.

```
1000 IF MEM-R*8<300 THEN DISP "Insufficient Mem" @ RETURN
1010 IF N<1 OR N>C+1 THEN DISP "Nonexistent Col" @ RETURN
1020 DISP "Working..." @ C=C+1 @ DIM A(R,C)
1030 G8=R @ G9=C-(C=N) @ T8=R @ T9=C
1040 FOR I=1 TO R @ T9=T9-1 @ IF NOT T9 THEN T9=C @ T8=T8-1
1050 NEXT I
1060 A(G8,G9)=A(T8,T9)
1070 T9=T9-1 @ IF NOT T9 THEN T9=C @ T8=T8-1
1080 G9=G9-1 @ IF NOT G9 THEN G9=C @ G8=G8-1
1090 IF NOT G8  THEN 1120
1100 IF G9=N THEN A(G8,G9)=0 @ GOTO 1080
1110 IF G8>0 AND T8>0 THEN 1060
1120 RETURN
```

Delete a column at N. Again, the data will be scrambled, so a shuffle occurs in a similar manner. First, the data is column shifted so that the column to be removed is the last one. Then the data is shifted down starting at the front and working up. The last locations in the array will be lost when the dimension statement is executed.

```
1000 IF N<1 OR N>C THEN DISP "Nonexistent Col" @ RETURN
1010 DISP "Working..." @ IF N=C THEN 1040
1020 FOR I=1 TO R @ FOR J=N TO C-1
1030 A(I,J)=A(I,J+1) @ NEXT J @ NEXT I
1040 G8=1 @ G9=1 @ T8=1 @ T9=1
1050 FOR I=1 TO R @ FOR J=1 TO C-1
1060 A(G8,G9)=A(T8,T9) @ G9=G9+1 @ IF G9>C THEN G9=1
     @ G8=G8+1
1070 T9=T9+1 @ IF T9>C THEN T9=1 @ T8=T8+1
1080 NEXT J @ T9=T9+1 @ IF T9>C THEN T9=1 @ T8=T8+1
1090 NEXT I @ C=C-1 @ DIM A(R,C) @ RETURN
```


Error Messages: MSG$ & Translator

SECTION 14

14.1 MSG$

The MSG$ keyword provides retrieval of error message text from the mainframe, plug-in modules, or LEX files. Each MSG$ LEX file should contain prompts and messages for an application program. This leaves a hook for foreign language translators to work with. The syntax for the keyword is:

MSG$(*numeric expression*)

The first three digits of the message number contain the LEX id, and the second three digits contain the message number. Leading zeros may be suppressed. As an example, suppose the 21st message of a LEX file id 94 is needed: A$=MSG$(94021).

The MSG$ keyword will work with translators. If a translator is present, MSG$(24) would return the same message as MSG$(1024) if a mainframe translator is present.

The heaviest use of MSG$ will be to display prompts, error messages and status messages in an application package. MSG$ used in this way allows customization for foreign languages. Keeping messages in a LEX file message table may also save ROM space. For example, if your LEX file number is 17, use

```
70 INPUT "",MSG$(17003);C$
```

instead of:

```
70 INPUT "",CHR$(27)&"<Color?"&CHR$(27)&">";C$
```

which will allow other language translators to handle the prompt. Other examples are provided in previous sections.

The MSG$ keyword is in LEX file 81. The use of MSG$ in a particular pac requires a LEX file with a built-in message table. This can be constructed using the HP-71 IDS volume I as a guide.

14.2 Translators

A translator is a LEX file whose sole purpose is to translate messages from the resident English to a foreign language. These LEX files are composed of tables and a poll handler which intercepts the pMEM, pERROR, pWARN, and pTRANS polls to substitute alternate message numbers.

The following convention has been set up to facilitate error trapping with language translators.

For mainframe messages:

Translated message number = ERRN+1000

For other LEX files:

Translated message number = ERRN+128

For example, mainframe error 57 is "File Not Found". If an ON ERROR routine is trapping for this error and must allow for foreign language messages, the appropriate statement is:

```
IF ERRN=57 OR ERRN=1057 THEN ....
```

The HP-IL error 255031 is "Directory Full". If an ON ERROR routine is trapping for this error and must allow for foreign language messages, the appropriate statement is:

```
IF ERRN=255031 OR ERRN=255159 THEN ....
```

This extended error trapping can be shortened with the user-defined function:

```
DEF FNE(X)= (X=ERRN) OR (X=ERRN+128+(X<1000)*872)
```

and the previous two examples above can be compressed to:

```
IF FNE(57) THEN ....
IF FNE(255031) THEN ....
```


Speed and Space

SECTION 15

The disadvantages of packing code need little enumeration: the risks are extreme. If packing must occur, caution is advised. If a working program is being packed in order to fit into available ROM space, we suggest that the author maintain a *very complete* audit trail. Some packing techniques actually improve speed as well, however combining code into user defined functions (DEF FNA(X)) can slow down the program, as additional time is required by the operating system to set up the call to the function. This slowdown can be up to .6 second for a function, and 1 second for a subprogram.

15.1 Variable Names

Single letter variable names save a byte for each reference, and slightly improve speed. Large groups of variables under one letter slow down the searching. For example, it would be better to use variables A, B, C, and D than C0, C1, C2, and C3.

15.2 Line References

When entering a label reference, such as GOTO HELP, don't enter the quotes. This will save a byte. The quote will appear on decompile. Remember: if you edit the line later on, use the [-CHAR] key to avoid re-entering quotes.

A GOTO pointing to a line that has a single letter label will save a byte as compared to using a GOTO pointing to a line number. This works best in instances where many GOTO statements refer to a single line.

Don't use GOTO after THEN or ELSE. Simply use the line number or or a label.

15.3 Multi-line Statements

Multi-line statements save two bytes for each line number saved.

15.4 Loops

FOR ... NEXT loops can be a source of speed improvement under some conditions. For instance, suppose each element in a 5 by 100 element array is to be incremented by 3. The following two blocks of code would do the same job, but the one on the right would execute faster.

```
100 FOR I=1 TO 100
110 FOR J=1 TO 5
120 A(I,J)=A(I,J)+3
130 NEXT J
140 NEXT I
```

```
100 FOR J=1 TO 5
110 FOR I=1 TO 100
120 A(I,J)=A(I,J)+3
130 NEXT I
140 NEXT J
```

The speed increase comes from the inner loop having less stack searching to perform for each NEXT statement.

15.5 Clearing Arrays and Strings

Numeric arrays may be cleared (all elements set to zero) very quickly by DESTROYing them and executing a new DIM statement. The operating system defaults all elements to zero.

In cases where a long string is to be set to spaces, a similar technique may be used. For instance, suppose a 100 character string of all spaces is needed:

```
2340 G$="" @ G$[100]=" "
```

The operating system will "pad" the missing characters from the beginning to 99 with spaces.

15.6 Logical Expressions

Logical expressions can be very useful in constructing numeric expressions, and generally save code. Logical expressions return a 1 or 0 depending on the evaluation of a comparison. For instance,

Use: `100 X=3-(Y=7 OR LEN(K$))`

Instead of: `100 IF Y=7 OR K$#"" THEN X=2 ELSE X=3`

15.7 Device Addressing

Addressing devices with the HP-IL module may be accomplished with a variety of commands. Generally, as the ease and luxury of the addressing mode increases, the amount of work the HP-71 has to do increases. The following table illustrates the relative times required to address a device as compared to other addressing methods.

METHOD	SPEED	
(:LOOP)	Fast	
<addr>	.	
%50	.	
DISPLAY	.	
HP82905B	.	
Volume Level	Slow	(Limited by media access times)

The fastest method of addressing a device is by its address on the loop. The loop will slow down as the number of devices present increases, and depending on the type of devices and their response times, the rate of increase in addressing times may be non-linear. A simple way to maximize the speed of addressing is to search once for the address of a device, and store that address in a variable for future use in the program. For example:

```
100 R=DEVADDR("HP82164A")      R = Address of RS-232 interface
   ...
1200 OUTPUT :R;T$ @ ENTER :R USING F$; I$
   ...
5480 OUTPUT :R;D$ @ RETURN
```

HPAF File Standard

SECTION 16

The Applications File format (HPAF), is intended to allow exchange of data between various programs. The format provides room for information that describes the structure of the data, so that various programs may make use of and exchange the data.

HPAF files are of type DATA, and may reside in either the HP-71 or a mass storage device, such as the HP82161A digital cassette drive.

The HPAF files are composed of three major sections: a *header*, the *data records* and an optional *descriptor block*. An example of such a file looks like this:

Rec #	Contents	Description
0	"HPAFNNS"	Type string: two numbers, one string
1	4	There are four records of data
2	12	The descriptor block starts at 12
3	77,9.3,"RED"	First data record
4	78,9.4,"BLUE"	..
5	81.5,10.3,"GREEN"	..
6	82.9,10.4,"GREEN"	Last data record
..		Empty data records
..		Empty data records
12	"COLNAMS",3,"TEMP" "VISCOSITY","COLOR" "DEGREES",1,"KELVIN"	Descriptor block

The following sections describe the header, the data records, and the descriptor block.

16.1 Header Information

The header must contain the following items:

1) Record zero contains a type string. The type string serves two purposes. The first four characters indicate the file is a HPAF file. The remaining characters describe the number of data items in each record, and their type. For example: HPAFNNS. The characters NNS indicate that there are three items in each record: the first two are numbers, and the third is a string.

2) Record 1 contains the number of data records that contain information. This number may be less than the total number of available records, allowing room for additional records to be added later, or the optional descriptor block.

3) Record 2 contains the address of the optional descriptor block. If no descriptor block is present, this number should be zero.

16.2 Data records

The data records begin in record 3, and must end before the descriptor block. Note that all data items for each record must fit within each logical record, so that any record may be accessed randomly. To compute the optimal logical record length for the file, remember that each number written in the record occupies 8 bytes, and each string occupies 3 bytes plus the number of bytes in the string. In addition, there must be one byte for the end of record mark. For example, if each record is going to hold two numbers and a ten character string, the record length must be at least 2*8+3+10+1, or 30 bytes. For more information about creating DATA files, see the HP-71 owner's manual, section 14.

16.3 Descriptor block

The descriptor block is optional. The descriptor block must come after the data records, and record 2 must contain the address of the first item in the block. Information in the descriptor block consists of *tags* which identify the type of information that follows, followed by the *number of items* associated with the tag, followed by the *items* themselves. The *tag* must be a string, the *number of items* must be a number, and the *items* must be strings. If numeric values are to be in the *items*, they should be string representations (STR$).

tag, number of items, item one [,item two...]

The information in the descriptor block may be written serially, or, if the logical record size is sufficiently large, written one tag to a record. In either case, the descriptor block must be able to be read serially

For example, to describe the names of the columns, a temperature offset, and the fact that the temperature units are degrees Kelvin, the descriptor block for the file might look like this:

Rec#	File contents	Comments
67)	"COLNAMS",3,"TEMP","VISCOSITY","DENSITY"	Column names
	"OFFSET",1,"2.172"	Offset
	"UNITS",1,"KELVIN"	Units information

(EOF)

SECTION 17

String Functions

The LEX file STRINGLX provides 11 keywords that enhance the string manipulation capabilities of the HP-71.

17.1 MEMBER

The MEMBER keyword returns the location of the first character in a subject string that is a member of a set string.

Syntax:

MEMBER(*subject string*,*set string* [,*starting position*])

Examples:

MEMBER(A$,"0123456789")
Returns the location of the first numeric character in A$.

MEMBER(A$,"0123456789",12)
Returns the location of 1st numeric character at/after position 12.

B=MEMBER(A$,B$,C

17.2 LTRIM$, RTRIM$, TRIM$

These keywords trim specified characters from the ends of string arguments. LTRIM$ trims characters from the left end, RTRIM$ trims characters from the right end, and TRIM$ trims characters from both ends.

Syntax:

LTRIM$(*string expression* [,*string expression*])

RTRIM$(*string expression* [,*string expression*])

TRIM$(*string expression* [,*string expression*])

The first string expression contains the string to be trimmed. The second, optional string expression specifies which character is to be trimmed, if found. Only the first character of the second string parameter is used. The default is to trim spaces.

Examples:

```
LTRIM$("   abcd ")        LTRIM$("hhhpeace on earth","h")
="abcd "                  ="peace on earth"

RTRIM$(" abce   ")        RTRIM$("peace on earthppp","p")
=" abce"                  ="peace on earth"

TRIM$("   abcd  ")        TRIM$("zzzpeace on earthz","z")
="abcd"                   ="peace on earth"

T$=TRIM$(G$)
```


17.3 LWC$, LWRC$

These keywords convert all uppercase characters in a string to their lowercase counterparts. The keywords are identical except in name.

Syntax:

LWC$(*string expression*)
LWRC$(*string expression*)

Examples:

```
LWC$("THIS IS NICE")
="this is nice"

A$="THIS IS NICE"
DISP A$
"this is nice"
```

17.4 REV$

This keyword reverses the order of the characters in a string.

Syntax:

REV$(*string expression*)

Examples:

```
REV$("2FBC3")
="3CBF2"

A$=REV$(B$)
```

```
REV$("palindrome")
="emordnilap"
```

An address stored in memory is backwards when obtained with a PEEK. REV$ is useful when converting the address into decimal:

```
DISP "The decimal address is";HTD(REV$(PEEK$("2F647",5)))
```


17.5 ROT$

This keyword rotates the contents of a string a specified number of places to the right. If the number of spaces is negative, the string will be rotated to the left.

Syntax:

ROT$(*string expression,numeric expression*)

Examples:

```
ROT$("12345",1)
="51234"

R$=ROT$(D$,5)
```

```
ROT$("12345",-1)
="23451"
```

17.6 RPT$

The RPT$ keyword concatenates multiple copies of a string expression together to form the resulting string.

Syntax:

RPT$(*string expression,numeric expression*)

Examples:

```
RPT$("X",4)
="XXXX"
```

```
RPT$("FRED",3)
="FREDFREDFRED"
```


17.7 SBIT

The SBIT keyword returns the value of a specific bit in a character string. It is most useful when analyzing the contents of the HP-71 graphics display.

Syntax:

SBIT(*string expression,numeric expression,numeric expression*)

The string expression is the string to be examined. The first numeric expression specifies which character to examine. The second numeric expression specifies which bit in the specified character to examine. Bits are numbered 0-7.

Examples:

`B=SBIT(GDISP$,1,0)`	Returns the bit value of the upper left pixel in the display.
`B=SBIT(A$,N,4)`	Returns the value of bit 4 in the Nth character of the string A$.

17.8 SBIT$

The SBIT$ keyword allows enhanced bit manipulation of data in strings.

Syntax:

SBIT$(*str exp, num exp [,num exp[,num exp]]*)

The first numeric expression specifies which byte in the string is to be modified. Other bytes in the string will be unchanged.

The second numeric expression specifies the bit to be manipulated. If not present, the byte specified by the first expression will be complemented. Bits are numbered 0-7.

The third numeric expression specifies the new value for the bit specified by the previous numeric expression. If not present, the bit will be complemented

Examples:

`A$=SBIT$(A$,5)`	Complement the fifth byte
`A$=SBIT$(A$,3,1)`	Complement the bit one of byte 3
`A$=SBIT$(A$,N,J,0)`	Clear bit J in byte N

17.9 SPAN

The SPAN keyword returns the location of first character found in a subject string that is not a member of a set string.

Syntax:

SPAN(*subject string,set string [,starting position]*)

Example:

`SPAN("123456e89","0123456789")`	Returns 7
`SPAN("123456x89r1.3","0123456789",8)`	Returns 10
`B=SPAN(A$,B$,C)`	

BREAKPT: BASIC Breakpoint System

SECTION 18

The BREAKPT program is a LEX file which provides breakpoint capability for debugging BASIC programs. When BREAKPT is in the HP-71, three new keywords become available: BREAK, UNBREAK, and BLIST. These keywords allow setting, clearing, and listing of breakpoints in BASIC program execution. Setting a breakpoint in this manner is equivalent to inserting a PAUSE statement at the beginning of a program line.

The BREAKPT program works by intercepting a poll each time a statement is executed. This will slow down an application program significantly, and should be used with caution in time sensitive situations.

BLIST

Lists all breakpoints in order of entry.

BREAK *<line number>* [,*<line number>* . . .]

Sets breakpoints at specified line numbers. Any number of breakpoints may be specified, separated by commas.

UNBREAK

Clears all breakpoints.

KEYBOARD IS – Using A Terminal

SECTION 19

The FORTH/Assembler ROM provides a set of keywords that permit keyboard entries to originate from devices on the HP-IL loop. These keywords are ESCAPE, KEYBOARD IS, and RESET ESCAPE. The KEYBOARD IS statement assigns one HP-IL device to act as a remote keyboard for the HP-71. The ESCAPE statement specifies that a particular one-character escape sequence received by the HP-71 from the current KEYBOARD IS device will be replaced by an HP-71 key code. This permits mapping of terminal-specific features to the HP-71 keyboard. The RESET ESCAPE statement clears out any existing mapping specified by ESCAPE statements. Refer to the FORTH/Assembler ROM Owner's Manual for a detailed discussion of these keywords.

19.1 KEYBOARD IS With HP-150

The following routine is useful when configuring an HP-150 as a remote keyboard and display device

```
10 IF POS(VER$,"KBD:")=0 THEN BEEP 1450,.08 @ DISP "Need KEYB
   OARD lex file!" @ END
20 RESET ESCAPE @ REAL A @ DIM E$ @ E$=CHR$(27)
30 'RS232WT': CLEAR :RS232 @ REMOTE @ OUTPUT :RS232 ;"SEO;SE3
   ;" @ LOCAL
40 A=SPOLL("rs232") @ IF A 929 THEN 'RS232WT' ELSE A=DEVADDR(
   "RS232")
50 ESCAPE "Q",105 @ ESCAPE "N",105 @ ESCAPE "R",105 ! I r
60 ESCAPE "i",103 ! Back
70 ESCAPE "D",47 @ ESCAPE "C",48 @ ESCAPE "A",50 @ ESCAPE "B"
   ,51
80 ESCAPE "p",43 @ ESCAPE "q",89 @ ESCAPE "r",150 @ ESCAPE "s
   ",109 ! ATTN, FETCH,Cmds,User
90 ESCAPE "t",162 @ ESCAPE "u",159 @ ESCAPE "v",160 @ ESCAPE
   "w",163 ! Top, <<<,>>>,Bottom
100 ESCAPE "h",102 @ ESCAPE "F",46 @ ESCAPE "J",107 @ ESCAPE
    "K",107 ! SST, Run,-Line,-Line
110 ESCAPE "@",92 ! Auto
120 OUTPUT :A ;E$&"&f1k0a16d2L  Attn    KON)  "&E$&"p";
130 OUTPUT :A ;E$&"&f2k0a16d2L FETCH           "&E$&"q";
140 OUTPUT :A ;E$&"&f3k0a16d2LCommand  Stack   "&E$&"r";
150 OUTPUT :A ;E$&"&f4k0a16d2L  User  (toggle)"&E$&"s";
160 OUTPUT :A ;E$&"&f5k0a16d2L  Top            "&E$&"t";
170 OUTPUT :A ;E$&"&f6k0a16d2L   Far    Left   "&E$&"u";
180 OUTPUT :A ;E$&"&f7k0a16d2L  Far    Right   "&E$&"v";
190 OUTPUT :A ;E$&"&f8k0a16d2L Bottom          "&E$&"w";
200 OUTPUT :A ;E$&"&s1A";E$&"&jB"; ! Set strap to emit escape
    sequences, User keys
```

```
210 LC OFF @ SFLAG -21
220 DISPLAY IS :RS232 @ KEYBOARD IS :RS232
```

19.2 KEYBOARD IS With HP-2648 Terminal

The following routine will configure an HP-2648 terminal as the remote keyboard. The terminal cursor keys are active, as are the insert/delete character keys. Pressing ESC twice gives the [ATTN] keystroke. [CTL][BACKSPACE] gives the [BACK] character. [f1] is [ATTN], [f2] is FETCH, [f3] is the command stack, [f4] is the user mode ID COMMANDS] is g[^], [f6] is g[<], [f7] is g[>], and [f8] is g[v].

The [CLEAR DSPLY] key also gives the -LINE command. The 'home' key recalls the first line of the current workfile, and [CTL] 'home' key recalls the last line of the current workfile.

```
10 RESET ESCAPE @ REAL A @ DIM E$ @ E$=CHR$(27)
20 'RS232WT': CLEAR :RS232 @ REMOTE @ OUTPUT :RS232 ;"SE0;SE
3;" @ LOCAL
30 A=SPOLL("rs232") @ IF A#929 THEN 'RS232WT' ELSE A=DEVADDR
("rs232")
40 ESCAPE "Q",105 @ ESCAPE "N",105 @ ESCAPE "R",105 ! I/r,I/
r,I r (to exit insert)
50 ESCAPE "i",103 ! Back
60 ESCAPE CHR$(27),43 ! Attn
70 ESCAPE "D",47 @ ESCAPE "C",48 @ ESCAPE "A",50 @ ESCAPE "B
",51 ! Left,Right,Up,Down
80 ESCAPE "p",43 @ ESCAPE "q",89 @ ESCAPE "r",150 @ ESCAPE "
s",109 ! Attn,FETCH,Cmds,User
90 ESCAPE "t",162 @ ESCAPE "u",159 @ ESCAPE "v",160 @ ESCAPE
 "w",163 ! Top,<<<,>>>,Bottom
100 ESCAPE "h",162 @ ESCAPE "F",163 ! Top, Bottom
110 ESCAPE "1",102 @ ESCAPE "2",46 @ ESCAPE "J",107 @ ESCAPE
 "K",107 ! Sst,Run,-Line,-Line
120 ESCAPE "#",92 ! Auto
130 OUTPUT :A ;E$&"&f1k0a2L"&E$&"p";
140 OUTPUT :A ;E$&"&f2k0a2L"&E$&"q";
150 OUTPUT :A ;E$&"&f3k0a2L"&E$&"r";
160 OUTPUT :A ;E$&"&f4k0a2L"&E$&"s";
170 OUTPUT :A ;E$&"&f5k0a2L"&E$&"t";
180 OUTPUT :A ;E$&"&f6k0a2L"&E$&"u";
190 OUTPUT :A ;E$&"&f7k0a2L"&E$&"v";
200 OUTPUT :A ;E$&"&f8k0a2L"&E$&"w";
210 OUTPUT :A ;E$&"&s1A"; ! Set strap to transmit escape seq
uences
220 LC OFF @ SFLAG -21
230 DISPLAY IS :RS232 @ KEYBOARD IS :RS232
```


19.3 Disabling KEYBOARD IS

Use the following routine when turning off the remote keyboard:

```
10 DISPLAY IS :DISPLAY @ KEYBOARD IS * @ RESET ESCAPE
20 CFLAG -21 @ RESET HPIL
```

Graphics

SECTION 20

The LCD display of the HP-71 may be used to depict graphic images using the GDISP statement. The contents of the LCD display may be read to a string with the GDISP$ statement. The HP-71 Owner's Manual (p. 137) has a discussion of these statements. Several tools are provided to assist in preparation of a graphics image. They are a graphics editor, a keyword PATTERN$, and the keywords SBIT and SBIT$, found in the STRINGLX file (see the chapter String Functions.)

20.1 GEDIT – Graphics Editor

The GEDIT program provides a facility for interactivly creating a graphics image on the LCD. To create an image, run GEDIT, and use redefined keys to move the cursor and set or clear points. The following keys are active when GEDIT is running:

```
[.]    Turn pixel on
[SPC]  Turn pixel off

[<]    Move cursor one pixel left
[>]    Move cursor one pixel right
[^]    Move cursor one pixel up
[v]    Move cursor one pixel down

[C]    Copy column, shifting display to right
[D]    Delete column, shifting display to left
[G]    Goto x,y location in display
[I]    Insert blank column
[L]    Display current location
[P]    Print graphic image on ThinkJet printer
[R]    Read image from file
[S]    Save image to file (as 132 character string)
[Q]    Exit program
```

```
10 ! GEDIT - Graphics Editor (Requires HP-IL keywords)
20 CALL GEDIT @ SUB GEDIT
30 DIM A$[132],B$[132]
40 DISP @ A$=GDISP$ @ X=1 @ Y=0 @ F8=FLAG(0) @ F9=FLAG(5)
50 GDISP A$
60 K$="" @ CFLAG 5
70 Z=FLAG(0,BIT(NUM(A$[X,X]),Y))
80 B$=A$ @ B$[X,X]=CHR$(BINEOR(NUM(B$[X,X]),2^Y))
90 ON TIMER #1,.3 GOSUB 290
100 K$=KEY$ @ IF K$="" THEN 100
110 F9=POS("#. SRIDCLGPQ",UPRC$(K$[1,1]))+1
120 ON F9 GOTO 50,150,130,140,190,200,210,220,230,240,260,280
    ,360
130 A$[X,X]=CHR$(BINIOR(NUM(A$[X,X]),2^Y)) @ GOTO 50
140 A$[X,X]=CHR$(BINAND(NUM(A$[X,X]),BINCMP(2^Y))) @ GOTO 50
150 IF K$="#47" THEN X=MOD(X-2,132)+1 @ GOTO 50
160 IF K$="#48" THEN X=MOD(X,132)+1 @ GOTO 50
170 IF K$="#50" THEN Y=MOD(Y-1,8) @ GOTO 50
180 IF K$="#51" THEN Y=MOD(Y+1,8) @ GOTO 50 ELSE 50
190 DISP "SAVE: "; @ GOSUB 330 @ PRINT #1,0;A$ @ GOTO 50
200 DISP "READ: "; @ GOSUB 330 @ READ #1,0;A$ @ GOTO 50
210 A$=A$[1,X-1]&CHR$(0)&A$[X,131] @ GOTO 50
220 A$=A$[1,X-1]&A$[X+1]&CHR$(0) @ GOTO 50
230 A$=A$[1,X-1]&A$[X,X]&A$[X,131] @ GOTO 50
240 DISP "X:";X;"   Y:";Y+1
250 IF KEYDOWN THEN 250 ELSE 50

270 X=MOD(INT(X-1),132)+1 @ Y=MOD(INT(Y-1),8) @ GOTO 50
280 PRINT CHR$(27)&"*132G";A$ @ GOTO 50
290 A=FLAG(5,NOT FLAG(5))
300 IF K$#"" THEN RETURN
310 IF A THEN GDISP B$ ELSE GDISP A$
320 RETURN
330 INPUT "File: ";F$
340 IF F$="" THEN POP @ GOTO 50
350 ASSIGN #1 TO F$ @ RETURN
360 CFLAG 0 @ DISP "Done" @ F8=FLAG(0,F8) @ F9=FLAG(5,F9) @
    END SUB
```


20.2 PATTERN$

The PATTERN$ keyword returns a character string which contains the GDISP$ equivalent of an ascii string in the display. The resulting string will contain 6 bytes for each character in the string argument

SYNTAX: PATTERN$(*string expression*)

EXAMPLE: GDISP PATTERN$("123")
A$=PATTERN$("Hello")

20.3 Example

A graphic image may be frozen on the left of the display with the WINDOW statement. Some applications may find this useful when implementing a user interface. In this example, a train is created in T$, placed in the display, and frozen in place for a prompt.

```
10 CALL GEX @ SUB GEX @ OPTION BASE 1
20 DIM T$[18] @ FOR I=1 TO 18 @ READ T @ T$[I]=CHR$(T) @ NEXT I
30 DATA 1,0,66,126,194,66,[illegible]8,200,201,72,73,206,200,[illegible]
   ,0,0
40 GDISP T$ @ WINDOW 4 @ INPUT "Destination?";D$
50 WINDOW 1 @ DISP "Going to ";D$ @ END SUB
```

Forth Utilities

SECTION 21

The following is a description of a collection of utilities developed to facilitate FORTH programming and debugging. There are five categories of words:

- **Decompiling:** UN: and RS.. These words are used to produce a map of a colon-compiled dictionary entry, and to decompile the contents of the return stack.
- **Single-stepping:** BP, BREAK, CONT, FINISH, STEP, and SST. These words are used to interrupt execution of a FORTH secondary word and single step each word or group of words.
- **Memory examination:** DUMP, DUMP+, LIST, ROOM?, S., and SHOW. These words are used to examine the contents of memory.
- **Output:** D-*, D-D, D-P, D-I, DELAY00, PAUSE, PRINT, and SKIP. These words are used to assign the display, pause during execution, and configure the printer.
- **Miscellaneous:** BASE?, TIME, and TIMED.

21.1 Loading FORTH Utility Files

There are three FORTH utility files: FTHUTILA, FTHUTILF, and FTHUTILC. If you have not established a FORTHRAM file, use the FTHUTILC file as follows:

```
COPY FTHUTILC[:TAPE] TO FORTHRAM
```

If you have already established FORTHRAM, the new words may be added with a two step procedure from within the FORTH environment:

```
"FTHUTILA[:TAPE]" ASSEMBLE

"FTHUTILF[:TAPE]" LOADF
```

The FTHUTILA file *must* be assembled first, as its words are subsequently used by words in the FTHUTILF file.

21.2 Decompiling

UN:	→

Decompile the word following UN: in the input stream. Used in the form:

UN: *<word name>*

UN: produces a complete map of a colon-compiled dictionary entry, showing the contents of the word header, and an addressed list of the words comprising the decompiled word's definition. For example, execution of

UN: ADDR- produces output like this:

```
Word: ADDR-
LFA: 30104      Link: 30093
NFA: 30109      5814444425UA
CFA: 30115      E701A :
     3011A      E0B3A 5-
     3011F      E0271 @
     30124      E71E8 ;
```

The first column of numbers show the address of each element of the word; the second column show the content of the address. After the CFA, the content is the CFA of a FORTH word, which is also identified by its name. From the above we can read off that the definition of ADDR- is : ADDR- 5- @ ; .

The rate at which UN: displays successive words in a definition is controlled by the PAUSELN variable.

UN: does not necessarily give a definition listing exactly the same as the original definition, because of the nature of certain common FORTH words. BEGIN and THEN, for example, have no compiled representations. UN: does allow you to determine the location of these structures by displaying the destination address for all branches. An IF word, for example, is displayed like this:

IF to *xxxxx*

where *xxxxx* is the address of the word that will be executed next if the flag tested by IF is false.

A second class of words for which the decompilation does not match the original definition exactly consists of words that are compiled as multiple words. Examples are OF and LEAVE. OF is compiled as OVER = IF DROP; LEAVE is compiled as R> R> 2DROP ELSE (the ELSE's here are just unconditional branches).

Finally, UN: does not recognize the headerless words used in the FORTH ROM dictionary, which may cause problems if you attempt to decompile a ROM word. In most cases, UN: will just display Unknown for a word it doesn't know. If the unknown word advances the instruction pointer when executed, UN: will get out of synch and produce garbage or hang up. The headerless words are listed in the FORTH IMS.

RS.	→

Decompiles the contents of the return stack. RS. lists each item on the return stack, in bottom-to-top order, each followed by the name of the word identified by the address. The lowest two levels, which refer to the outer interpreter, are omitted.

21.3 Single Stepping

The words STEP, SST, BREAK, BP, CONT, and FINISH enable you to interrupt a FORTH secondary word at any point in its execution and single step each successive word or group of words in its definition. A separate return-stack and instruction pointer environment is set up for the word, so that you can carry out various FORTH operations between steps, and so that return-stack operations included in the word will not confuse the normal outer interpreter. The interrupted word uses the normal data stack, so that any operations you perform between steps must leave the stack in the state expected by the next step.

Interrupted execution of a word XXX is initiated by either STEP *XXX* or *<address>* BREAK *XXX*. Both methods set up the interrupt environment, then begin executing *XXX*. STEP executes only the first word (after the " : ") in *XXX*'s definition. BREAK executes *XXX* up to *<address>*, or to the final " ; ", whichever is encountered first.

Execution of an interrupted word is resumed by the words SST, CONT, and FINISH. SST executes the next word in the definition; CONT resumes continuous execution, stopping at the next encounter of the breakpoint address (which can be reset with BP), or at the end of the word. FINISH clears any breakpoint setting and completes execution of the word through the final " ; ".

Each time a word is interrupted, a user-selectable vectored word is executed. The CFA of the vectored word is stored in the variable SSTOUT. The default SSTOUT word is S., which displays the stack in bottom-to-top order (reverse of .S) with in square brackets [ ].

Single stepping proceeds through a word's definition at the level of the definition - each secondary in the definition is executed entirely as a single step. SST does not wander up and down through the various levels of secondaries in a definition. BREAK and CONT will stop at a breakpoint address set at any level, but a subsequent SST will halt back at the top-level of the original word's definition. You can effectively single-step through lower levels by setting breakpoints in the low level definition and using CONT.

The single-step words use two user variables during their execution. #2FB7F is used to pass the address of the start of the single step environment to the single step primitives. #2FB84, which is also used by the colon compiler, is used to hold the current breakpoint address. FORTH words that are tested with BREAK or SST must not disturb the contents of these variables. Furthermore, they must not disturb the return stack pointer stored in RP0, nor move the return stack itself. In particular, do *not* BREAK or SST words containing GROW or SHRINK at any level.

BP	*n* →

Set a breakpoint at address *n*, for use with CONT.

BREAK *n* →

Used in the form <*addr*> BREAK <*wordname*>.

Create a single step environment for the word named next, then execute the word, stopping when the instruction pointer reaches the address on the top of the data stack. The addresses for BREAK can be obtained using UN: on the word to be single stepped. BREAK can stop at any word address in a definition after the first address following the : (use STEP if you want to stop on the first address) and before the final ; (stopping on the ; is the same as executing the full word).

CONT →

Resume execution of a word that was interrupted with STEP, BREAK, or SST. Execute up to the breakpoint address, or to the final " ; ", whichever comes first.

FINISH →

Complete execution of an interrupted word through the final " ; ".

SST →

Display the name of the word identified by the next address in the current single-step word's definition, then execute the named word. Then execute the word whose CFA is stored in the variable SSTOUT. The default SSTOUT word is S. which displays the stack in bottom-to-top order (reverse of .S).

STEP →

Used in the form STEP <*wordname*>.

Create a single step environment for the word named next, then SST the first word following the : in the word's definition.

21.4 Memory Examination

DUMP *addr n* →

Display *n* nibbles, starting at *addr*, as ASCII hex characters.

DUMP+ *addr n* → *addr+n*

Display *n* nibbles, starting at *addr*, as ASCII hex characters. Leave the next address (*addr+n*) on the stack.

LIST →

Display a list of user-dictionary words, starting with the most recently created.

ROOM? →

Display the number of nibbles available in the FORTHRAM file.

S. →

Display the data stack contents, in bottom first, top last order (opposite of .S), inside [ ] brackets.

SHOW → *addr+5n*

Display the address and contents of *n* consecutive 5-nibble cells, starting at *addr*. Leave the next address on the stack. Display time is controlled by PAUSELEN.

21.5 Output

D-* →

Execute "DISPLAY IS *" BASICX

D-D →

Execute "DISPLAY IS DISPLAY" BASICX

D-P →

Execute "DISPLAY IS PRINTER" BASICX

D-R →

Execute "DISPLAY IS RS232" BASICX

DELAY00 →

Execute "DELAY 0,0" BASICX

PAUSE →

Pause for the number of milliseconds stored in the variable PAUSELEN. (Does an empty DO LOOP). Intended for use with outputs to the HP-71 display.

PAUSELEN → *addr*

Return the address of the variable containing the delay in milliseconds produced by PAUSE.

PRINT →

Used in the form PRINT xxxxx, which causes the display output of the FORTH word xxxxx to be directed to the printer (:PRINTER(1)). The original DISPLAY IS device is restored automatically after xxxxx has finished execution. PRINT UN: FRED, for example, will print the decompilation of FRED on a printer instead of the display.

SKIP →

Send ESC &l1L to the :PRINTER to set perforation skip mode.

21.6 Miscellaneous

BASE? →

Display the current base in decimal.

TIME →

Pushes the current HP-71 clock time onto the floating point stack. Time is expressed in seconds from midnight, rounded to the nearest .01 second.

TIMED →

Used in the form TIMED xxxx, which displays the execution time of the word xxxx in seconds (to the nearest .01 second). For timing floating point words, be aware that TIMED will change the L-register on input, and the T- and Z- registers on output.